Chapter FIVE

Multi-threading

Introduction

One of the powerful features of Java is its built-in support for multithreading—the concurrent running of multiple tasks within a program.

E.g., A Web browser is an example of a multithreaded application. Within a typical browser, you can scroll a page while it's downloading an applet or an image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish.

Multithreaded programs extend the idea of multitasking by taking it one level: individual programs will appear to do multiple tasks at the same time. Each task is usually called a thread—which is short for thread of control. Programs that can run more than one thread at once are said to be multithreaded. Multithreading: is a process of executing multiple threads simultaneously. Java Multithreading is mostly used in games, animation etc.

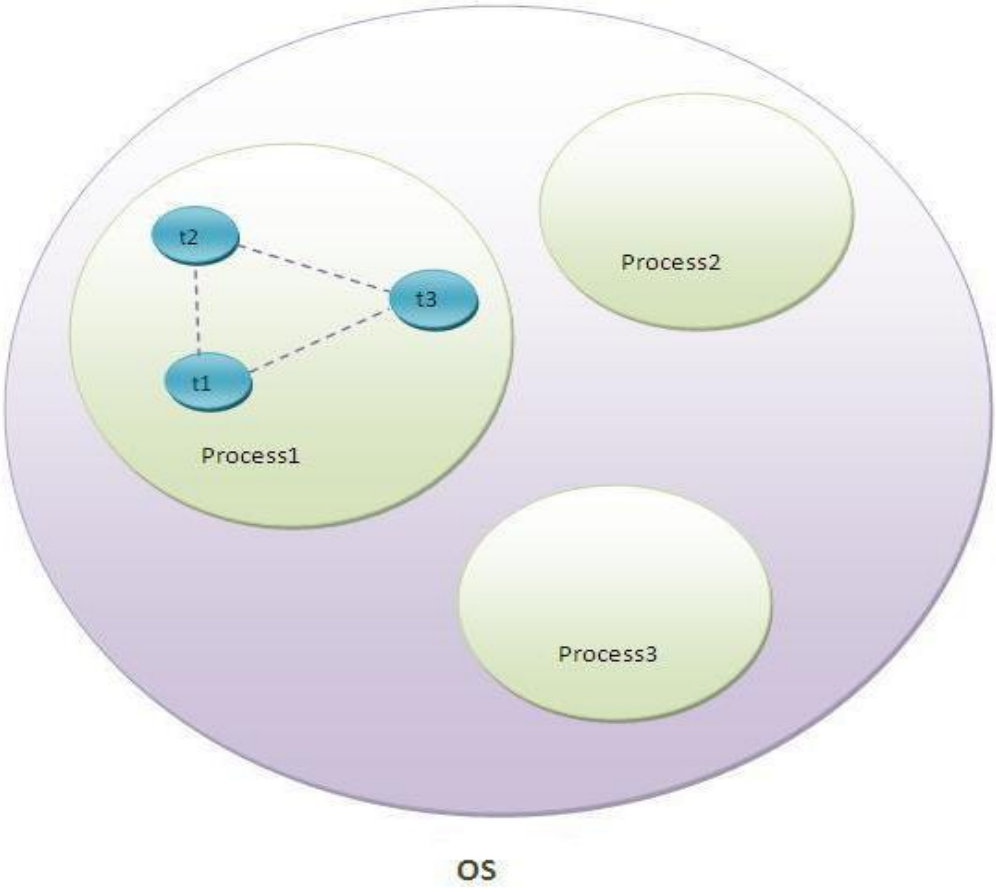
Advantage of Java Multithreading

It doesn't block the user because threads are independent and you can perform multiple operations at same time. You can perform many operations together so it saves time. Threads are independent so it doesn't affect other threads if exception occur in a single thread.

What is Thread?

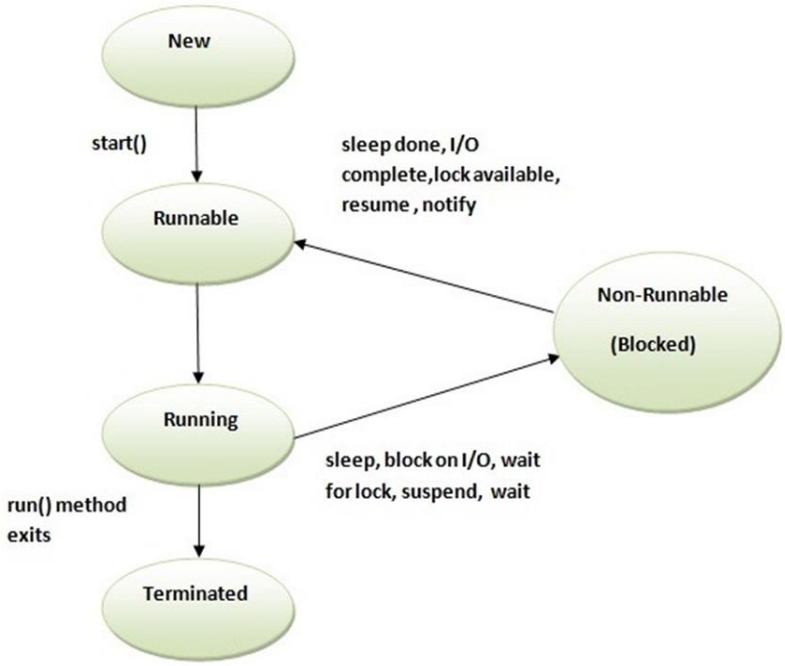
A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area. As shown in the figure below, thread is executed inside the process. There is context-switching between the threads. Context switching is the process of storing and restoring of CPU state so that thread execution can be resumed from the same point at a later point of time. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time one thread is executed only.



Life cycle of a Thread (Thread States)

1) New
2) Runnable
3) Running
4) Non-Runnable (Blocked)
5) Terminated



1. New: The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. Runnable: The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. Running: The thread is in running state if the thread scheduler has selected it.
4. Non-Runnable (Blocked): This is the state when the thread is still alive, but is currently not eligible to run.
5. Terminated: A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

1. **Thread class:** Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

   Constructors of Thread class

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

Commonly used methods of Thread class

1. public void run(): is used to perform action for a thread.
2. public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
3. public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. public void join(): waits for a thread to die.
5. public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
6. public int getPriority(): returns the priority of the thread.
7. public int setPriority(int priority): changes the priority of the thread.
8. public String getName(): returns the name of the thread.
9. public void setName(String name): changes the name of the thread.
10. public Thread currentThread(): returns the reference of currently executing thread.
11. public int getId(): returns the id of the thread.
12. public Thread.State getState(): returns the state of the thread.
13. public boolean isAlive(): tests if the thread is alive.
14. public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. public void suspend(): is used to suspend the thread(depricated).
16. public void resume(): is used to resume the suspended thread(depricated).
17. public void stop(): is used to stop the thread(depricated).

Example:

```
class Multi extends Thread{

public void run(){
```

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start(); } }

Output:thread is running...

2. Runnable interface: The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Example

```
class Multi implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi m1=new Multi();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks: A new thread starts (with new callstack). The thread moves from new state to the Runnable state. When the thread gets a chance to execute, its target run() method will run.

Thread Scheduler

Thread scheduler in java is the part of the JVM that decides which thread should run. There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process. The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a. higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks.The scheduler then determines which task should execute next, based on priority and other factors.

Sleep method

sleep() method is used to sleep a thread for the specified milliseconds of time.The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example:

```
class TestSleepMethod1 extends Thread{

 public void run(){

  for(int i=1;i<5;i++){

    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}

    System.out.println(i);

  }  }

 public static void main(String args[]){

  TestSleepMethod1 t1=new TestSleepMethod1();

  TestSleepMethod1 t2=new TestSleepMethod1();
```

```
t1.start();

t2.start();

}

}
```

Joining with threads

The Thread API contains a method for waiting for another thread to complete: the join() method. When you call Thread.join(), the calling thread will block until the target thread completes. Thread.join() is generally used by programs that use threads to partition large problems into smaller ones, giving each thread a piece of the problem.

Syntax:

- public void join()throws InterruptedException
- public void join(long milliseconds)throws InterruptedException

```
// A Java program for understanding

// the joining of threads

// import statement

import java.io.*;

// The ThreadJoin class is the child class of the class Thread

class ThreadJoin extends Thread  {

// overriding the run method

public void run()  {

for (int j = 0; j < 2; j++)  {

try  {
```

```
// sleeping the thread for 300 milli seconds

Thread.sleep(300);

System.out.println("The current thread name is: " + Thread.currentThread().getName());

}

// catch block for catching the raised exception

catch(Exception e)  {

System.out.println("The exception has been caught: " + e);

}

System.out.println( j );

}  }  }

public class ThreadJoinExample

{

// main method

public static void main (String argvs[])

{

// creating 3 threads

ThreadJoin th1 = new ThreadJoin();

ThreadJoin th2 = new ThreadJoin();

ThreadJoin th3 = new ThreadJoin();

// thread th1 starts

th1.start();
```

```
// starting the second thread after when
// the first thread th1 has ended or died.
try  {
System.out.println("The current thread name is: "+ Thread.currentThread().getName());
// invoking the join() method
th1.join();
}
// catch block for catching the raised exception
catch(Exception e)  {
System.out.println("The exception has been caught " + e);
}
// thread th2 starts
th2.start();
// starting the th3 thread after when the thread th2 has ended or died.
try  {
System.out.println("The current thread name is: " + Thread.currentThread().getName());
th2.join();
}
// catch block for catching the raised exception
catch(Exception e)  {
System.out.println("The exception has been caught " + e);
```

}

// thread th3 starts

th3.start(); } }

Output:

```
The current thread name is: main
The current thread name is: Thread - 0
0
The current thread name is: Thread - 0
1
The current thread name is: main
The current thread name is: Thread - 1
0
The current thread name is: Thread - 1
1
The current thread name is: Thread - 2
0
The current thread name is: Thread - 2
1
```

Example: join(long miliseconds) Method Example

Filename: **TestJoinMethod2.jav**

class TestJoinMethod2 extends Thread{

public void run(){

for(int i=1;i<=5;i++){

try{

Thread.sleep(500);

}catch(Exception e){System.out.println(e);}

System.out.println(i);

} }

public static void main(String args[]){

```
TestJoinMethod2 t1=new TestJoinMethod2();

TestJoinMethod2 t2=new TestJoinMethod2();

TestJoinMethod2 t3=new TestJoinMethod2();

t1.start();

try{

 t1.join(1500);

}catch(Exception e){System.out.println(e);}

t2.start();

t3.start();     }    }
```

Output:

```
 1
2
3
1
4
1
2
5
2
3
3
4
4
5
5
```

Thread Priority

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

FileName: **ThreadPriorityExample.java**

```
// Importing the required classes

import java.lang.*;

public class ThreadPriorityExample extends Thread

{

// Method 1

// Whenever the start() method is called by a thread

// the run() method is invoked

public void run()
```

```
{
// the print statement
System.out.println("Inside the run() method");
}
// the main method
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();
// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value
// 1st Thread
// Displaying the priority of the thread
// using the getPriority() method
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
// 2nd Thread
// Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// 3rd Thread
```

```
// // Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// Setting priorities of above threads by
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);
// 6
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
// 3
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// 9
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
// Main thread
// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
// Priority of the main thread is 10 now
Thread.currentThread().setPriority(10);
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
} }
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

Example:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

Thread Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization.

The Java language provides two keywords for ensuring that data can be shared between threads in a controlled manner: synchronized and volatile.

```
synchronized(object)

{

// statements to be synchronized

}
```

Synchronized has two important meanings:

- it ensures that only one thread executes a protected section of code at one time (mutual exclusion or mutex), and
- it ensures that data changed by one thread is visible to other threads (visibility of changes).

Without synchronization, it is easy for data to be left in an inconsistent state. Synchronization allows us to define blocks of code that must run atomically, in which they appear to execute in an all-or-nothing manner, as far as other threads can tell.

Example

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
```

```
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```